DSA Tutorial Week 3 - Trees and Search

Problem 0 - Preparation

You are given the following set of numbers:

```
15, 23, 5, 7, 9, 25, 19, 17
```

a. Construct a binary search tree (BST) using these numbers. Begin by inserting them into the BST in the order given. Illustrate the process using pen and paper, showing the tree after each insertion.

b. Once your tree is constructed, perform the following operations:
   i. Insert an additional number, 13, into your BST and redraw the tree.
   ii. Delete a number of your choice from the BST and redraw the tree.
   iii. Perform an in-order traversal of the BST and list the resulting sequence of numbers.

c. After completing these operations, focus on binary search:
   i. Choose three numbers from your final tree: one that exists in the tree, one that does not, and the root number.
   ii. Using *binary search*, describe the process of searching for each of these numbers in your BST. Illustrate each step of the process.

d. Create an Array from BST: Convert your final BST into a sorted array. An in-order traversal of the BST will give you the elements in ascending order. Record this sorted array.
   i. Choose the same three numbers that you used for the binary search on the BST (one present in the tree, one absent, and the root number).
   ii. For each of these numbers, perform a binary search on the extracted array. Document each step of your search process, noting the middle element you compare with and the sub-array (or sub-section of the array) you choose to continue the search in after each comparison.

To verify your work and gain additional insights, use the online tool visualgo (https://visualgo.net/en/bst). Compare the tool's output with your own work to understand any discrepancies and to solidify your comprehension of BSTs and binary search.

Problem 1 - Binary Tree Maximum Path Sum

Write an algorithm to find the *maximum path sum* in a binary tree. A path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain *at least one node* and *does not need to go through the root*.

Example:

Consider the following binary tree:

```
    10
   /  \
  2   10
```

```
 / \    \
20  1   -25
        /  \
       3    4
```

In this tree, some of the possible paths and their sums are:

Path: 20 → 2 → 10 → 10 Sum = 42
Path: 1 → 2 → 10 Sum = 13
Path: 10 → 10 Sum = 20
Path: 3 → -25 → 4 Sum = -18
Path: 3 → -25 → 10 Sum = -12

However, the maximum path sum for this tree is 42, which is achieved by the path 20 → 2 → 10 → 10.

1. Start by discussing the problem and finding an algorithm in pseudocode.
   How will it traverse the tree? What steps are needed?
   Consider the elements of the recursion, how does it branch, what does it return, what is the base-case?
2. When you have a grasp of the algorithm, implement the function `maxPathSum(TreeNode root)` in the attached **BinaryTreeMaximumPathSum.java** that returns the maximum path sum.
3. The function should use recursion to traverse the tree and calculate the sum of each possible path.
4. At each node, consider the maximum sum obtained by including that node and either of its left or right subtrees, or both.
5. The maximum path sum for a node should be the maximum of:
   a. The node's value.
   b. Node's value + maximum path sum of the left subtree.
   c. Node's value + maximum path sum of the right subtree.
   d. Node's value + maximum path sum of both subtrees.
6. Keep track of the global maximum path sum during the traversal.

TEACHER POINTERS

- Make sure to maintain a global state (a class-level variable or a reference) that keeps track of the maximum path sum found at any node in the tree.
- The path can include a sequence of nodes going up the tree and then down another branch.
- Negative values in nodes can affect the maximum sum.
- Local vs. Global Maxima: At each node, consider two sums:
  - **Local Maximum:** The maximum sum including the current node and either of its subtrees (left or right), but not both. This sum is relevant for parent nodes to consider.
  - **Global Maximum:** The highest path sum found so far in the tree. This sum may include the current node and both its subtrees. This is the final answer but isn't always passed up the recursive calls.
- If a subtree's maximum sum is negative, it's sometimes better not to include that subtree in the path. Remember that a path can consist of a single node. Think about the special case in which we do want to include negative numbers.
- At each node, update the global maximum if the sum of the current node value and the maximum sum of the left and right subtrees is greater than the current global maximum.

- In your recursive function, the base case should handle null nodes, typically returning 0, as they do not contribute to the path sum.
- If you're not getting the expected output, try adding print statements in your recursive function to see the path sums being calculated at each step.

EXAMPLE SOLUTION

```
function maxPathSum(root):
    if root is null:
        return 0

    leftMax = max(0, maxPathSum(root.left))
    rightMax = max(0, maxPathSum(root.right))
    maxSum = root.val + leftMax + rightMax

    // update globalMax with maxSum if it's greater
    return root.val + max(leftMax, rightMax)
```

Also, refer to the attached `BinaryTreeMaximumPathSum.java` in the “Answers” directory for the full solution in Java.

Problem 2 - Receiver Roster[2]

Coach Bell E. is trying to figure out which of her football receivers to put in her starting lineup. In each game, Coach Bell wants to start the receivers who have the highest performance (the average number of points made in the games they have played), but has been having trouble because her data is incomplete, though interns do often add or correct data from old and new games. Each receiver is identified with a unique positive integer jersey number, and each game is identified with a unique integer time.

Describe a data structure supporting the following operations, each in ***worst-case O(log n)*** time, where n is the number of games in the database at the time of the operation. Assume that n is always larger than the number of receivers on the team.

`record(g, r, p)`	record **p** points for jersey **r** in game **g**
`clear(g, r)`	remove any record that jersey **r** played in game **g**
`ranked_receiver(k)`	return the jersey with the $\mathbf{k}^{\mathbf{th}}$ highest performance

There's no need to implement the methods, the goal is to find and design an appropriate data structure.

1. Design a data structure that meets the operation requirements.
2. Explain your choices and how they contribute to the efficiency of the system.
3. Discuss the potential challenges and how your design addresses them.

Pointers

- How should each node represent games and the performance of various receivers? Consider the information each node must store.
- How will you store data for each game and corresponding receiver performances? Think about efficient access and update of this data.
- A receiver's performance is an average across multiple games. How will you efficiently track and update this?
- What's the mechanism for inserting or updating a receiver's performance in a game? How does each operation affect the overall data structure?
- When recording or clearing data, how will you ensure the receiver's average performance is accurately updated?
- How can you efficiently determine the jersey number of the receiver with the $\boldsymbol{k^{th}}$ highest performance? Consider how to track and sort performances.
- What primary and auxiliary data structures will you use? How will they interact with each other?

EXAMPLE SOLUTION

First, observe that operations require finding and modifying records for a receiver given a jersey number in ***O(log n)*** time, so maintain a dictionary containing the receivers keyed by unique jersey numbers (let's call it the jersey dictionary). Since we need to achieve worst-case ***O(log n)*** running time, we cannot afford the ***expected performance*** of a hash table, so we implement the dictionary using a Set implemented as an AVL tree.

For each receiver, we will need to find and update their games by game ID, so for each receiver in the jersey dictionary, we will maintain a pointer to their ***own Set AVL tree*** containing that receiver's games

[2] Source: https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_prob4sol/

keyed by game ID (call this a receiver's game dictionary). With each receiver's game dictionary, we will maintain the number of games they've played and the total number of points they've scored to date. We can compare the performance of two jerseys from their respective number of games and points via cross multiplication.

Lastly, to find the ***kth*** highest performing receiver, we maintain a ***separate Set AVL*** tree on the receivers keyed by performance, augmenting each node with the size of its subtree (call this the performance tree). We showed in lecture how to maintain subtree size in **O(1)** time, so we can maintain this augmentation. Each node of the jersey dictionary will store a cross-linking pointer to the node in the performance tree corresponding to that player. Since we use Set AVL trees for all data structures, Set operations run in worst-case **O(log n)** time.

Jersey Dictionary Node:

```
JerseyNode {
    int jerseyNumber;
    ReceiverNode receiverNode; // Pointer to Receiver's Game Dictionary
    PerformanceNode performanceNode; // Pointer to node in Performance Tree
}
```

Receiver's Game Dictionary Node:

```
ReceiverNode {
    int gameID;
    int pointsScored;
    int totalPoints; // Aggregated at root of the tree
    int numberOfGames; // Aggregated at root of the tree
}
```

Performance Tree Node:

```
PerformanceNode {
    float performance; // Calculated as totalPoints / numberOfGames
    int jerseyNumber;
    int subtreeSize; // Size of subtree rooted at this node
}
```

To implement **`record(g, r, p)`**, find player r's game dictionary D in the receiver dictionary in ***O(log n)*** time. If game g is in D, update its stored points to p in ***O(log n)*** time and update the total number of points stored with r's game dictionary in ***O(1)*** time. Otherwise, insert the record of game **g** into **D** in ***O(log n)*** time, and update the number of games and total points stored in ***O(1)*** time. The performance of r may have changed, so find the node corresponding to r in the performance tree, remove the receiver's performance from the tree, update its performance, and then reinsert into the tree all in **O(log n)** time. This operation maintains the semantics of our data structures in worst-case **O(log n)** time.

```
Function record(g, r, p):
    if GameTree contains g:
        update GameTree[g] with new points p
    else:
        insert new game g into GameTree
    update ReceiverTree[r] with new total points and games
```

```
    update PerformanceTree with new average for r
```

To implement **`clear(g, r)`**, find player **r**'s game dictionary **D** in the receiver dictionary as before and remove **g** (assuming it exists). Identically to above, maintain the stored number of games and total points, and update the performance tree together in worst-case ***O(log n)*** time.

```
Function clear(g, r):
    if GameTree contains g:
        remove game g from GameTree
        update ReceiverTree[r] with adjusted points and games
        update PerformanceTree with new average for r
```

To implement ranked **`receiver(k)`**, find the kth highest performance in the performance tree by using the subtree size augmentation: if the size of a node's right subtree is k or larger, recursively find in the right subtree; if the size of the node's right subtree is ***k − 1***, then return the jersey stored at the current node; otherwise the size of the node's right subtree is $k_0 < k-1$, recurse in the node's left subtree to find its subtree's k_0^{th} highest performing player. This recursive algorithm only walks down the tree, so it runs in worst-case ***O(log n)*** time

More detailed implementations

record:

```
Function record(int g, int r, int p) {
    JerseyNode jersey = jerseyDictionary.find(r);
    ReceiverNode receiver = jersey.receiverNode;

    if (receiver.containsGame(g)) {
        receiver.updateGame(g, p);
    } else {
        receiver.addGame(g, p);
    }

    updatePerformanceTree(jersey.performanceNode, receiver);
}

Function updatePerformanceTree(PerformanceNode node, ReceiverNode receiver) {
    performanceTree.remove(node);
    float newPerformance = calculatePerformance(receiver);
    performanceTree.insert(newPerformance, receiver.jerseyNumber);
}
```

clear:

```
Function clear(int g, int r) {
    JerseyNode jersey = jerseyDictionary.find(r);
    ReceiverNode receiver = jersey.receiverNode;

    if (receiver.containsGame(g)) {
        receiver.removeGame(g);
        updatePerformanceTree(jersey.performanceNode, receiver);
    }
```

```
}
```

ranked_receiver:

```
Function ranked_receiver(int k) {
    return findKthHighest(performanceTree, k);
}

Function findKthHighest(PerformanceNode node, int k) {
    if (node == null) return null;

    int rightTreeSize = node.right ? node.right.subtreeSize : 0;

    if (k <= rightTreeSize) {
        return findKthHighest(node.right, k);
    } else if (k == rightTreeSize + 1) {
        return node.jerseyNumber;
    } else {
        return findKthHighest(node.left, k - rightTreeSize - 1);
    }
}
```

Problem 3 - Find Prime[3]

Consider a binary tree containing **N** integer keys whose values are all less than **K**,
and the following `Find-Prime` algorithm that operates on this tree. The algorithm and supporting functions are given below. Your task is to analyze their ***time complexity***.

```
Function Find-Prime(T )
      x = Tree-Min(T )
      while x ≠ nil
            x = Tree-Successor(x)
            if Is-Prime(x.key)
            return x
      return x

Function Is-Prime(n)
      i = 2
      while i · i ≤ n
            if i divides n
                  return false
            i = i + 1
      return true
```

```
Function Tree-Successor(x)
      if x.right ≠ nil
            return Tree-Minimum(x.right)
      y = x.parent
      while y ≠ nil and x == y.right
            x = y
            y = y.parent
      return y

Function Tree-Minimum(x)
      while x.left ≠ nil
            x = x.left
      return x
```

Pointers

1. **Understand Each Function:** Begin by understanding what each function does: `Find-Prime` searches for prime numbers in a binary tree, `Is-Prime` checks if a number is prime, `Tree-Successor` finds the next node in in-order traversal, and `Tree-Minimum` finds the smallest node.
2. First analyze the functions' **individual time complexities**, starting with `Is-Prime`. Next, determine the complexity of `Tree-Minimum` and `Tree-Successor`. Consider the structure of the binary tree, particularly its height h.
3. `Find-Prime` calls `Tree-Successor` repeatedly and might call `Is-Prime` for each node. **Combine their complexities** to find the overall complexity of `Find-Prime`.
4. Consider the **best** and **worst-case** scenarios for the tree's structure (e.g., a balanced tree vs. a skewed tree).
5. Write down the ***final complexity*** and explain how you arrived at it.

Next, use the provided ***TreeNode.java*** and ***BinarySearchTree.java*** and implement the `find-prime, is-prime, tree-successor` and `tree-minimum` functions in Java.

TEACHER POINTERS

Encourage Step-by-Step Analysis:

Prompt students to first understand each function individually before combining the complexities. Focus on the traversal and prime-checking aspects separately.

Discuss Binary Tree Properties:

[3] Source: https://www.inf.usi.ch/carzaniga/edu/algo19s/exercises.pdf

Explain how the height of the tree (h) impacts the complexity, especially in different types of trees (e.g., balanced vs. skewed trees).

Prime Number Concept:

Ensure students understand the concept of prime numbers and why the Is-Prime function's complexity is O(√n).

Complexity Combination:

Guide students through combining the complexities of the two operations. Discuss why we multiply these complexities instead of adding them.

Visual Aids:

Use tree diagrams to visually demonstrate the traversal process and how it affects complexity.

EXAMPLE SOLUTION

Analysis of Tree Traversal:

- Tree-Minimum Function:
  - Finds the minimum value in the tree.
  - Time Complexity: O(h), where h is the height of the tree.

- Tree-Successor Function:
  - Called iteratively to traverse the tree in in-order.
  - In the worst case, visits all N nodes in the tree.
  - Each call's Time Complexity: O(h).
  - Total Traversal Time Complexity: O(N * h).

Analysis of Prime Checking:

- Is-Prime Function:
  - Checks if a number n is prime by iterating from 2 up to the square root of n.
  - Time Complexity: O(√n).
- Applying to Keys in Tree:
  - Each key's value is less than K.
  - Time Complexity for each key: O(√K).

Combining the Complexities:

- Overall Find-Prime Algorithm:
  - Integrates tree traversal and prime checking.
  - For each of the N keys, a prime check is potentially performed.
  - Worst-Case Time Complexity: O(N * h * √K), where:
    - N is the number of nodes in the tree.
    - h is the height of the tree.
    - K is the upper bound on the value of the keys.